# Chapter 1
# Advances in Large-scale RDF Data Management

Peter Boncz, Orri Erling, Minh-Duc Pham

**Abstract** One of the prime goal of the LOD2 project is improving the performance and scalability of RDF storage solutions so that the increasing amount of Linked Open Data (LOD) can be efficiently managed. Virtuoso has been chosen as the basic RDF store for the LOD2 project, and during the project it has been significantly improved by incorporating advanced relational database techniques from MonetDB and Vectorwise, turning it into a compressed column store with vectored execution. This has reduced the performance gap ("RDF tax") between Virtuoso's SQL and SPARQL query performance in a way that still respects the "schema last" nature of RDF. However, by lacking schema information, RDF database systems such as Virtuoso still cannot use advanced relational storage optimizations such as table partitioning or clustered indexes and have to execute SPARQL queries with many self-joins to a triple table, which leads to more join effort than needed in SQL systems. In this chapter, we first discuss the new column store techniques applied to Virtuoso, the enhancements in its cluster parallel version, and show its performance using the popular BSBM benchmark at the unsurpassed scale of 150 billion triples. We finally describe ongoing work in deriving an "emergent" relational schema from RDF data which, can help to close the performance gap between relational-based and RDF-based storage solutions.

## 1.1 General Objectives

One of the objectives of the LOD2 EU project is to boost the performance and the scalability of RDF storage solutions so that it can, efficiently manage huge datasets (e.g., one trillion RDF triples) of Linked Open Data (LOD). However, it has been

Peter Boncz, Minh-Duc Pham
CWI, Amsterdam e-mail: {P.Boncz,duc}@cwi.nl

Orri Erling
OpenLink Software, U.K. e-mail: erling@xs4all.nl

noted that given similar data management tasks, relational database technology still significantly outperforms RDF data stores. One controlled scenario in which the two technologies can be compared is the BSBM benchmark [5], which exists equivalent relational and RDF variants. As illustrated in Figure 1.1, while the SQL systems can process by up to 40-175K QMpH, the Triple stores can only reach 1-10K QMpH, showing a factor of 15-40 of performances difference.

In the LOD2 project we investigated the causes of this large difference (the "RDF tax", i.e. the performance cost of choosing RDF instead of relational database technology). Here we identify three causes:

Table 12. Performance for multiple clients, 25M dataset (in QMpH).

| Dataset size 25M | Number of clients | | | | |
|---|---|---|---|---|---|
| | 1 | 2 | 4 | 8 | 64 |
| Sesame | 1,343 | 1,485 | 1,204 | 1,300 | 1,271 |
| Jena TDB | 353 | 513 | 694 | 536 | 555 |
| Jena SDB | 968 | 1,346 | 1,021 | 883 | 927 |
| Virtuoso TS | 4,123 | 7,610 | 9,491 | 5,901 | 5,400 |
| D2R Server | 140 | 187 | 160 | 146 | 143 |
| MySQL | *18,578* | *31,093* | *39,647* | *40,599* | *40,470* |
| Virtuoso SQL | *69,585* | *85,146* | *135,097* | *173,665* | *148,813* |

**Fig. 1.1** Triple Stores vs. SQL: a heavy "RDF Tax" (2009)

- the particular case of BSBM is to the disadvantage of RDF as BSBM by its nature is very relational: its schema has just few classes, and all its properties occur exactly once for each subject, such that the data structure is very tabular. As such, the ease of use of SPARQL to formulate queries on irregularly structured data does not come int play, and the complications to which such irregular data leads in relational solutions (many more tables, and many more joins) are avoided.
- relational systems are quite mature in their implementations. For instance, the explore workload is an OLTP workload which relational systems target with key index structures and pre-compiled PL/SQL procedures. The BI workload of BSBM benefits from analytical execution techniques like columnar storage and vectorized execution and hash joins with bloom filters (to name just a few). While relational products over the years have implemented many such optimizations, RDF stores typically have not. Another area where analytical relational database engines have made important progress is the use of cluster technology. Whereas in 1990s only the very high end of RDBMS solutions was cluster-enabled (i.e. Teradata), many other systems have been added such as Greenplum, Paraccel, Vertica, SAP HANA and SQLserver Parallel data Warehouse (which without exceptional also leverage columnar storage and vectorized or JIT-compiled execution).
- RDF stores do not require a schema but also do not exploit it, even though the structure of the data in fact is highly regular. This hurts in particular in the very common SPARQL star-patterns, which need to be executed using multiple self-joins, where relational systems do not need joins at all. The structure that *is* heavily present in RDF triples further leads to the co-occurrence of properties to be heavily (anti-)correlated. The complexity of query optimization is not only exponential with respect to the amount of joins (and SPARQL needs many more than SQL) but also relies on cost models, yet cost models typically become very unreliable in the face of correlations. Unreliable cost models lead to bad query plans and this very strongly affects performance and scalability of RDF stores. In all, query optimization for SPARQL is both more costly and unreliable than for SQL.

Virtuoso6, a high-performance RDF Quad Store, was chosen as the main RDF store for LOD2 knowledge base at the start of the project. In order to reduce the "RDF tax", we first revised architectural ideas from the state-of-the-art of relational database systems, particularly, advanced column stores such as MonetDB [2], Vectorwise [17]. Then, we brought some of the unique technologies and architectural principles from these column stores into Virtuoso7, making it work more efficiently on modern hardware. These techniques include tuning the access patterns of database queries to be CPU-cache conscious, and also making query processing amendable to deeply pipelined CPUs with SIMD instructions by introducing concepts like vector processing. We note that the insights gained in improving Virtuoso will also be useful for other RDF store providers to enhance their respective technologies as well.

Further, the cluster capabilities of Virtuoso were significantly improved. Note that by lack of table structures, RDF systems must distribute data by the triple (not tuple), which leads to more network communication during query execution. Network communication cost tends to be the limiting factor for parallel database systems, hence Virtuoso7 Cluster Edition introduced an innovative control flow framework that is able to hide network cost as much as possible behind CPU computation.

By the end of the LOD2 project, these improvements in Virtuoso7 on the BSBM BI workload strongly improved performance. A quantified comparison is hard as Virtuoso6 would not even complete the workload ("infinitely better" would be an exaggeration). Still, when comparing the SQL with the SPARQL implementation of BSBM-BI on Virtuoso7, we still see an "RDF tax" of a factor 2.5. This performance difference comes from the schema-last approach of RDF model: SPARQL plans need more joins than SQL and often the query plan is not optimal. To address this issue, CWI performed research in the line of the second bullet point above: the goal would be to give the RDF store more insight in the actual structure of RDF, such that SPARQL query plans need less self-joins and query optimization becomes more reliable. This goal should be achieved without losing the schema-last feature of RDF: there should be no need for an explicit user-articulated schema.

The idea of recovering automatically an "emergent" schema of actual RDF data is that RDF data in practice is quite regular and structured. This was observed in the proposal to make SPARQL query optimization more reliable by recognizing "characteristics sets" [10]. A characteristic set is a combination of properties that typically co-occur with the same subject. The work in [10] found that this number is limited to a few thousand on even the most complex LOD datasets (like DBpedia), and the CWI research on emergent schema detection that started in the LOD2 project [14] aims to further reduce the amount of characteristic sets to the point that characteristics sets become tables in a table of limited size (less than 100), i.e. further reducing the size. To that, the additional challenge of finding human-understandable labels (names) for tables, columns, and relationships was added. The goal of emergent schemata thus became two-fold: (1) to inform SPARQL systems of the schema of the data such that they need less self-joins and query optimization becomes more reliable, (2) to offer a fully relational view of an RDF dataset to end-users so that existing SQL applications can be leveraged on any RDF dataset. The latter goal

could help to increase RDF adoption and further help make relational systems more semantic (because all tables, columns and relationships in an emergent schema are identified by URIs).

In all, this chapter shows tangible progress in reducing the "RDF tax" and a promising avenue to further reduce the performance gap between SPARQL and SQL systems and even some hope of making them converge.

## 1.2 Virtuoso Column Store

The objective of the Virtuoso7 column store release was to incorporate the state of the art in relational analytics oriented databases into Virtuoso, specifically for the use with RDF data. Ideally, the same column store engine would excel in both relational analytics and RDF.

Having RDF as a driving use case emphasizes different requirements from a purely relational analytics use case, as follows:

**Indexed access.** Some column stores geared purely towards relational analytics [17] obtain excellent benchmark scores without any use of index lookup, relying on merge and hash join alone. An RDF workload will inevitably have to support small lookups without setup cost, for which indexed access is essential.

**Runtime data types.** Runtime data typing is fundamental to RDF. There must be a natively supported any data type that can function as a key part in an index. Dictionary encoding all literal values is not an option, since short data types such all numbers and dates must be stored inline inside an index. This offers native collation order and avoids a lookup to a dictionary table, e.g. before doing arithmetic or range comparisons. This is a requirement for any attempt at near parity with schema-first systems. Thus dictionary encoding is retained only for strings.

**Multi-part and partitionable indices.** Many early column stores [2] were based on an implicit row number as a primary key. Quite often this would be dense and would not have to be materialized. Such a synthetic row number is however ill suited to an RDF use case that further must be scale-out capable. Thus, Virtuoso does not have any concept of row number but rather has multi-part sorted column-wise compressed indices for all persistent data structure. In this, Virtuoso most resembles [8, 4]. This structure is scale-out friendly since partitioning can be determined by any high cardinality key part and no global synthetic row number needs to exist, even as an abstraction.

**Adaptive compression.** Column stores are renowned for providing excellent data compression. This comes from having all the values in a column physically next to each other. This means that values, even in a runtime typed system, tend to be of the same type and to have similar values. This is specially so for key parts where values are ascending or at least locally ascending for non-first key parts. However, since a single column (specially for the RDF object column) will store all the object values of all triples, there can be no predeclared hints for type or compression. Different parts of the column will have radically different data types, data ordering and

number of distinct values. Thus local environment is the only indication available for deciding on compression type.

**Transactionality.** While a column-wise format is known to excel for read-intensive workloads and append-style insert, RDF, which trends to index quads at least from S to O and the reverse. Thus with one index there can be mostly ascending insert but there is no guarantee of order on the other index. Also row-level locking needs to be supported for transactional applications. This is by and large not an OLTP workload but short transactions must still be efficient. Thus Virtuoso departs from the typical read-intensive optimization of column stores that have a separate delta structure periodically merged into a read-only column-wise compressed format [17, 8]. Virtuoso updates column-wise compressed data in place, keeping locks positionally and associating rollback information to the lock when appropriate. Thus there is no unpredictable latency having to do with flushing a write optimized delta structure to the main data. While doing so, Virtuoso has excellent random insert performance, in excess of the best offered by Virtuoso's earlier row store.

As a result of these requirements, Virtuoso uses a sparse row-wise index for each column-wise stored index. The row wise index is a B-Tree with one row for anywhere between 2000 to 16000 rows. This entry is called the row-wise leaf. To each row-wise leaf corresponds a segment of each column in the table. Each segment has an equal number of values in each column. Consecutive segments tend to be of similar size. When values are inserted into a segment, the segment will split after reaching a certain size, leading to the insertion of a new row-wise leaf row. This may cause the row-wise leaf page to split and so forth. Each column segment is comprised of one or more compression entries. A compression entry is a sequence of consecutive values of one column that share the same compression. The compression entry types are chosen based on the data among the following:

- **Run length.** Value and repeat count
- **Array.** Consecutive fixed length (32/64 nbit) or length prefixed strings
- **Run length delta.** One starting value followed by offsets and repeat counts for each offset
- **Bitmap.** For unique closely spaced integers, there is a start value and a one bit for each consecutive value, the bit position gives the offset from start value
- **Int delta.** From a start value, array of 16 bit offsets
- **Dictionary.** For low cardinality columns, there is a homogenous or heterogenous array of values followed by an array of indices into the array. Depending on the distinct values, the index is 4 or 8 bits.

Using the Virtuoso [7] default index scheme with two covering indices (PSOG and POSG) plus 3 distinct projections (SP, OP, GS), we obtain excellent compression for many different RDF datasets. The values are in bytes per quad across all the five indices, excluding dictionary space for string literals.

- BSBM: 6 bytes
- DBpedia: 9 bytes
- Uniprot: 6 bytes
- Sindice crawl: 14 bytes

DBpedia has highly irregular data types within the same property and many very differently sized properties, thus it compresses less. Uniprot and BSBM are highly regular and compress very well. The web crawl consists of hundreds of millions of graphs of 20 triples each, thus the graph column is highly irregular and thus less compressible, accounting for the larger space consumption. However one typically does not reference the graph column in queries, so it does not take up RAM at runtime.

### *1.2.1 Vectored Execution*

A column store is nearly always associated with bulk operators in query execution, from the operator at a time approach of MonetDB [2] to vectored execution [17, 8, 1]. The idea is to eliminate query interpretation overhead by passing many tuples between operators in a query execution pipeline. Virtuoso is no exception, but it can also run vectored plans for row-wise structures. The main operators are index lookup and different variations of hash, from hash join to group by. An index lookup receives a vector of key values, sorts them, does a log(n) index lookup for the first and subsequently knows that all future matches will be to the right of the first match. If there is good locality, an index lookup is indistinguishable from a merge join. The added efficiency of vectoring is relative to the density of hits. Considering that over a million rows are typically under one row-wise leaf page (e.g. 16K rows per segment * 70 segments per page), there is a high likelihood that the next hit is within the next 16K or at least within the next 1M rows, hence there is no need to restart the index lookup from the tree top.

Hash based operations use a memory-only linear hash table. This is essentially the same hash table for hash join, group by and distinct. The hash table consists of a prime number of fixed size arrays that are aligned by CPU cache line. Different fields of a 64 bit hash number give the array and a place in the array. The entry is either found at this place or within the same cache line or is not in the hash table. If a cache line is full and the entry does not match, there is an extra exceptions list which is typically short. One can determine the absence of a value with most often one cache miss. Only if the line is full does one need to consult the exceptions, leading to a second cache miss. Since the hash operations are all vectored, prefetch is used to miss multiple consecutive locations in parallel. Further, if the hash table contains pointers to entries instead of single fixed length integers, the high bits of the pointer are used to contain a filed of the hash number. Thus one does not dereference a pointer to data unless the high bits of the pointer (not used for addressing) match the high bits of the hash number. In this way cache misses are minimized and each thread can issue large numbers of cache misses in parallel without blocking on any.

Further, Bloom filters are used for selective hash joins. We have found that a setting of 8 bits per value with 4 bits set gives the best selectivity. Typically the Bloom filter drops most of non-matching lookup keys before even getting to the hash table.

### 1.2.2 Vector Optimizations

Virtuoso can adjust the vector size at runtime in order to improve locality of reference in a sparse index lookup. Easily 30% of performance can be gained if looking for 1M instead of 10K consecutive values . This comes from higher density of hits in index lookup. The vector size is adaptively set in function of available memory and actually observed hit density.

### 1.2.3 Query Optimization

All the advanced execution techniques described so far amount to nothing if the query plan is not right. During the last year of LOD2 we have made a TPC-H implementation to ensure that all state of the art query optimization techniques are present and correctly applied. TPC-H is not an RDF workload but offers an excellent checklist of almost all execution and optimization tricks [6].

The goal of LOD2 is RDF to SQL parity but such parity is illusory unless the SQL it is being compared to is on the level with the best. Therefore having a good TPC-H implementation is a guarantee of relevance plus opens the possibility of Virtuoso applications outside of the RDF space. Details are discussed in [3].

In the following we will cover the central query optimization principles in Virtuoso.

**Sampling.** Virtuoso does not rely on up-front statistics gathering. Instead, the optimizer uses the literals in queries to sample the database. The results of sampling are remembered for subsequent use. In RDF, there is an indexed access path for everything. Thus if leading P, S or O are given, the optimizer can just look at how many hits there in the index. The hits, if numerous, do not have to be counted. Counting the number of hits per page and number of pages is accurate enough. Also, within each RDF predicate, there is a count of occurrences of the predicate, of distinct S's, distinct O's and G's. These allow estimating the fan-out of the predicate, e.g. a *foaf:Name* has one O per S and *foaf:knows* has 100 O's per S. Also we recognize low cardinality properties, e.g. there is one city per person but 1M persons per city.

The statistics interact with runtime support of inference. Thus in one inference context, if tag is a super-property of about and mentions, but there are no triples with tag, the statistics automatically drill down to the sub-properties and sum these up for the super-property. This is however scoped to the inference context.

There can be conditions on dependent part columns, e.g. if P, S and G are given, G is likely a dependent part since in PSOG there is O between the leading parts and G. Thus sampling is used to determine the frequency of a specific G within a fixed P, S. The same is done for relational tables where there in fact are dependent columns that do not participate in ordering the table.

**Cost Model.** It has been recently argued [16] that SPARQL can be optimized just as well or even better without a cost model. We do not agree with this due to the following: It is true that a cost model has many complexities and possibilities for

error. However, there are things that only a cost model can provide, in specific, informed decision on join type.

There is a definite break-even point between hash join and vectored index lookup. This is tied to the input counts on either side of the join. Both the number of rows on the build and the probe sides must be known in order to decide whether to use hash join. Also, when building the hash table, one has to put as many restrictions as possible on the build side, including restrictive joins. To get this right, a cost model of one kind or another is indispensable. The choice hinges on quantities, not on the structure of the query. If the goal is only to do look-ups efficiently, then one can probably do without a cost model. But here the goal is to match or surpass the best, hence a cost model, also for RDF is necessary even though it is very complex and has a high cost of maintenance. It is also nearly impossible to teach people how to maintain a cost model. Regardless of these factors, we believe that one is indispensable for our level of ambition.

### *1.2.4 State of the RDF Tax*

We refer to the performance difference between a relational and RDF implementation of a workload as the RDF tax. This has been accurately measured with the Star Schema Benchmark [12], a simplified derivative of TPC-H. While Virtuoso does TPC-H in SQL [3] on a par with the best, the RDF translation of all the query optimization logic is not yet complete, hence we will look at SSB.

SSB has one large fact table (*lineorder*) and several smaller dimension tables (*part*, *supplier*, *dw_date*, *nation* and *region*). The schema is denormalized into a simple star shape. Its RDF translation is trivial; each primary key of each table is a URI, each column is a property and each foreign key is a URI.

SSB was run at 30G scale on a single server with Virtuoso, MonetDB and MySQL. In SQL, Virtuoso beats MonetDB by a factor of 2 and MySQL by a factor of 300 (see Table 1.1). In SPARQL, Virtuoso came 10-20% behind MonetDB but still 100x ahead of MySQL. These results place the RDF tax at about 2.5x in query execution time. Thanks to Virtuoso's excellent query performance, SPARQL in Virtuoso will outperform any but the best RDBMS's in analytics even when these are running SQL.

All plans consist of a scan of the fact table with selective hash joins against dimension tables followed by a simple aggregation or a group by with relatively few groups, e.g. YEAR, NATION. In the RDF variant, the fact table scan becomes a scan of a property from start to end, with the object, usually a foreign key, used for probing a hash table built from a dimension table. The next operation is typically a lookup on another property where the S is given by the first and the O must again satisfy a condition, like being in a hash table.

The RDF tax consists of the fact that the second column must be looked up by a self join instead of being on the same row with the previous column. This is the best case for the RDF tax, as the execution is identical in all other respects. There are

| Query | 30GB | | | | | 300GB | | |
|---|---|---|---|---|---|---|---|---|
| | Virtuoso SQL | Virtuoso SPARQL | RDF tax | MonetDB | MySQL | Virtuoso SQL | Virtuoso SPARQL | RDF tax |
| Q1 | 0.413 | 1.101 | 2.67 | 1.659 | 82.477 | 2.285 | 7.767 | 3.40 |
| Q2 | 0.282 | 0.416 | 1.48 | 0.5 | 74.436 | 1.53 | 3.535 | 2.31 |
| Q3 | 0.253 | 0.295 | 1.17 | 0.494 | 75.411 | 1.237 | 1.457 | 1.18 |
| Q4 | 0.828 | 2.484 | 3.00 | 0.958 | 226.604 | 3.459 | 6.978 | 2.02 |
| Q5 | 0.837 | 1.915 | 2.29 | 0.648 | 222.782 | 3.065 | 8.71 | 2.84 |
| Q6 | 0.419 | 1.813 | 4.33 | 0.541 | 219.656 | 2.901 | 8.454 | 2.91 |
| Q7 | 1.062 | 2.33 | 2.19 | 5.658 | 237.73 | 5.733 | 15.939 | 2.78 |
| Q8 | 0.617 | 2.182 | 3.54 | 0.521 | 194.918 | 2.267 | 6.759 | 2.98 |
| Q9 | 0.547 | 1.29 | 2.36 | 0.381 | 186.112 | 1.773 | 4.217 | 2.38 |
| Q10 | 0.499 | 0.639 | 1.28 | 0.37 | 186.123 | 1.44 | 4.342 | 3.02 |
| Q11 | 1.132 | 2.142 | 1.89 | 2.76 | 241.045 | 5.031 | 12.608 | 2.51 |
| Q12 | 0.863 | 3.77 | 4.37 | 2.127 | 241.439 | 4.464 | 15.497 | 3.47 |
| Q13 | 0.653 | 1.612 | 2.47 | 1.005 | 202.817 | 2.807 | 4.467 | 1.59 |
| Total | 8.405 | 21.989 | 2.62 | 17.622 | 2391.55 | 37.992 | 100.73 | 2.65 |

**Table 1.1** Star Schema Benchmark with scales 30GB and 300GB (in seconds)

some string comparisons, e.g. brand contains a string but these are put on the build side of a hash join and are not run on much data.

In a broader context, the RDF tax has the following components:

**Self-joins.** If there are conditions on more than one column, every next one must be fetched via a join. This is usually local and ordered but still worse than getting another column. In a column store, predicates on a scan can be dynamically reordered based on their observed performance. In RDF this would alter the join order and is not readily feasible.

**Cardinality estimation.** In a multi-column table one can sample several predicates worth in one go, in RDF this requires doing joins in the cost model and is harder. Errors in cost estimation build up over many joins. Accurate choice of hash vs index based join requires reliable counts on either side. In SQL analytics, indices are often not even present, hence the join type decision is self-evident.

**Optimization search space.** A usage pattern of queries with tens of triple patterns actually hitting only a few thousand triples leads to compilation dominating execution times. A full exploration of all join orders is infeasible, as this is in the order of factorial of the count of tables and there can easily be 30 or 40 tables. Reuse of plans when the plans differ only in literals is a possibility and has been tried. This is beneficial in cases but still needs to revalidate if the cardinality estimates still hold with the new literal values. Exploring plans with many joins pushed to the build side of a hash join further expands the search space.

**String operations.** Since RDF is indexed many ways and arbitrary strings are allowed everywhere, implementations store unique strings in a dictionary and a specially tagged reference to the dictionary in the index. Going to the dictionary makes a scan with a *LIKE* condition extremely bad, specially if each string is distinct. Use of a full text index is therefore common.

**URI's.** For applications that do lookups, as most RDF applications do, translating identifiers to their external, usually very long, string form is an RDF-only penalty. This can be alleviated by doing this as late as possible but specially string conditions on URI's are disastrous for performance.
**Indexing everything.** Since there is usually an indexed access path for everything, space and time are consumed for this. TPC-H 100G loads in SQL in 15 minutes with no indices other than the primary keys. The 1:1 RDF translation takes 12 hours. This is the worst case of the RDF tax but is limited to bulk load. Update intensive OLTP applications where this would be worse still are generally not done in RDF. Of course nobody forces one to index everything but this adds complexities to query optimization for cases where the predicate is not known at compile time.
**Runtime data typing.** This is a relatively light penalty since with vectored execution it is possible to detect a homogenous vector at runtime and use a typed data format. If a property is all integers, these can be summed by an integer-specific function. This usually works since RDF often comes from relational sources. DBpedia is maybe an exception with very dirty data, but then it is not large, hence the penalty stays small.
**Lack of schema.** There is usually no schema or the data does not comply with it. Therefore optimizations like late projection that are safe in SQL are not readily applicable. If you take the 10 youngest people and return their birth date, name and address you cannot defer getting the name and address after the top 10 since there might be people with many names or no names etc. These special cases complicate the matter but optimizations having to do with top k order are still possible. Similarly dependencies inside grouping columns in a group by cannot be exploited because one does not know that these are in fact functional even if the schema claims so.

Many of these penalties fall away when leaving the triple table format and actually making physical tables with columns for single valued properties. The exceptions may still be stored as triples/quads, so this does not represent a return to schema-first. Physical design, such as storing the same data in multiple orders becomes now possible since data that are alike occupy their own table. Also n:m relationships with attributes can be efficiently stored in a table with a multi-part key while still making this look like triples.

This is further analyzed in a later section of this chapter. Implementation in Virtuoso is planned for the summer of 2014.

## 1.3 Virtuoso Cluster Parallel

Virtuoso's scale out capability has been significantly upgraded during LOD2. The advances are as follows:
**Elastic partitions.** The data is sharded in a large number of self-contained partitions. These partitions are divided among a number of database server processes and can migrate between them. Usually each process should have one partition per hardware thread. Queries are parallelized to have at most one thread per partition.

Partitions may split when growing a cluster. Statistics are kept per partition for detecting hot spots.

**Free-form recursion between partitions.** One can write stored procedures that execute inside a partition and recursively call themselves in another partition, ad infinitum. This is scheduled without deadlocking or running out of threads. If a procedure waits for its descendant and the descendant needs to execute something in the waiting procedure's partition, the thread of the waiting procedure is taken over. In this way a distributed call graph never runs out of threads but still can execute at full platform parallelism. Such procedures can be transparently called from queries as any SQL procedures, the engine does the partitioning and function shipping transparently.

**Better vectoring and faster partitioning.** Even the non-vectored Virtuoso cluster combined data for several tuples in messages, thus implementing a sort of vectoring at the level of interconnect while running scalar inside the nodes. Now that everything is vectored, the architecture is simpler and more efficient.

**More parallel control flows.** The basic query execution unit in cluster is a series of cross partition joins, called DFG (distributed fragment). Each set of co-located joins forms a stage of the DFG pipeline. Each stage runs one thread per partition if there is work to do in the partition. The results are partitioned again and sent onwards. The DFG ends by returning vectors of query variable bindings to the query coordinator or by feeding them in an aggregation. An aggregation itself will be partitioned on the highest cardinality grouping key if the cardinality is high enough. A subsequent DFG can pick the results of a previous partitioned aggregation and process these through more joins again with full platform utilization.

**Different parallel hash joins.** Tables are usually partitioned and in the case of RDF always partitioned. However, if a hash join build side is small, it is practical to replicate this into every server process. In this way, what would be a non-colocated join from foreign key to primary key becomes colocated because the hash table goes to its user. However, if the probe key is also the partitioning key of the probe, there is never a need to replicate because the hash table can be partitioned to be colocated with the probe without replicating. If the hash table would be large but the probe key is not the partitioning key of the probing operator, the hash table can still be partitioned. This will require a message exchange (a DFG stage). However, this is scalable since each server will only host a fraction of the whole hash table. Selective hash joins have Bloom filters. Since the Bloom filter is much smaller than the hash table itself, it can be replicated on all nodes even if the hash table is not. This allows most of the selectivity to take place before the inter-partition message exchange (DFG stage).

With the embarrassingly parallel SSB, cluster shows linear throughput gains: 10x the data takes 5x longer on twice the hardware (See Table 1.1). This is the case for either RDF or SQL. The RDF tax is the same for cluster as for single server, as one would expect.

### *1.3.1 Performance Dynamics*

Running complex queries such as the BSBM BI workload makes high use of cross partition joins (DFG) and of nested subqueries. This is a DFG inside a DFG, where the innermost DFG must run to completion before the invoking stage of the calling DFG can proceed. An existence test containing a non-colocated set of joins is an example of such pattern.

We find that message scheduling that must keep track of distributed dependencies between computations becomes a performance bottleneck. Messages can be relatively fragmented and numerous. Scheduling a message involves a critical section that can become a bottleneck. In subsequent work this critical section has been further split. The scheduling itself is complex since it needs to know which threads are waiting for which operations and whether a descendant operation ought to take over the parent's thread or get its own.

All the techniques and observed dynamics apply identically to RDF and SQL but are worse in RDF because of more joins. Use of hash joins and flattening of subqueries alleviates many of these problems. Hash joins can save messages by replicating the hash table, so there are messages only when building the hash table. In a good query plan this is done on far less data than probing the hash table.

### *1.3.2 Subsequent Development*

Virtuoso is at present an excellent SQL column store. This is the prerequisite for giving RDF performance that is comparable with the best in relational data warehousing.

The next major step is storing RDF in tables when regular structure is present. This will be based on the CWI research, described in the next section. Query plans can be made as for triples but many self-joins can be consolidated at run time in into a table lookup when the situation allows. Cost model reliability will also be enhanced since this will know about tables and can treat them as such.

## 1.4 BSBM Benchmark Results

The BSBM (Berlin SPARQL Benchmark) was developed in 2008 as one of the first open source and publicly available benchmarks for comparing the performance of storage systems that expose SPARQL endpoints such as Native RDF stores, Named Graph stores, etc. The benchmark is built around an e-commerce use case, where a set of products is offered by different vendors and consumers have posted reviews about products. BSBM has been improved over this time and is current on release 3.1 which includes both Explore and Business Intelligence use case query mixes, the latter stress-testing the SPARQL1.1 group-by and aggregation function-

ality, demonstrating the use of SPARQL in complex analytical queries. To show the performance of Virtuoso cluster version, we present BSBM results [5] on the V3.1 specification, including both the Explore (transactional) and Business Intelligence (analytical) workloads (See the full BSBM V3.1 results for all other systems[1]).

We note that, comparing to the previously reported BSBM report[2] for 200M triples dataset, this BSBM experiment against 50 and 150 billion triple datasets on a clustered server architecture represents a major step (750 times more data) in the evolution of this benchmark.

### *1.4.1 Cluster Configuration*

RDF systems strongly benefit from having the working set of the data in RAM. As such, the ideal cluster architecture for RDF systems uses cluster nodes with relatively large memories. For this reason, we selected the CWI scilens[3] cluster for these experiments. This cluster is designed for high I/O bandwidth, and consists of multiple layers of machines. In order to get large amounts of RAM, we used only its "bricks" layer, which contains its most powerful machines. Virtuoso V7 Column Store Cluster Edition was set up on 8 Linux machines. Each machine has two CPUs (8 cores and hyperthreading, running at 2GHz) of the Sandy Bridge architecture, coupled with 256GB RAM and three magnetic hard drives (SATA) in RAID 0 (180 MB/s sequential throughput). The machines were connected by Mellanox MCX353A-QCBT ConnectX3 VPI HCA card (QDR IB 40Gb/s and 10GigE) through an InfiniScale IV QDR InfiniBand Switch (Mellanox MIS5025Q). The cluster setups have 2 processes per machine, 1 for each CPU. A CPU here has its own memory controller which makes it a NUMA node. CPU affinity is set so that each server process has one core dedicated to the cluster traffic reading thread (i.e. dedicated to network communication) and the other cores of the NUMA node are shared by the remaining threads. The reason for this set-up is that communication tasks should be handled with high-priority, because failure to handle messages delays all threads. These experiments have been conducted over many months, in parallel to the Virtuoso V7 Column Store Cluster Edition software getting ready for release. Large part of the effort spent was in resolving problems and tuning the software.

### *1.4.2 Bulk Loading RDF*

The original BSBM data generator was a single-threaded program. Generating 150B triples with it would have taken weeks. We modified the data generator to be able

[1] http://bit.ly/ZHtG5D

[2] http://bit.ly/12DpjMU

[3] This cluster is equipped with more-than-average I/O resources, achieving an Amdahl number $> 1$. See `www.scilens.org`.

to generate only a subset of the dataset. By executing the BSBM data generator in parallel on different machines, each generating a different part of the dataset, BSBM data generation now has become scalable. In these experiments we generated 1000 data files with the BSBM data generator. Separate file generation is done using the `nof` option in the BSBM driver. These files are then distributed to each machine according to the modulo of 8 (i.e., the number of machine) so that files number 1, 9, 17, ... go to machine 1, file number 2, 10, 18,... go to machine 2, and so on. This striping of the data across the nodes ensures a uniform load, such that all nodes get an equal amount of similar data.

| nr triples | Size (.ttl) | Size (.gz) | Database Size | Load Time |
|---|---|---|---|---|
| 50 Billion | 2.8 TB | 240 GB | 1.8 TB | 6h 28m |
| 150 Billion | 8.5 TB | 728 GB | 5.6 TB | n/a |

**Table 1.2** BSBM data size and loading statistic

Each machine loaded its local set of files (125 files), using the standard parallel bulk-load mechanism of Virtuoso. This means that multiple files are read at the same time by the multiple cores of each CPU. The best performance was obtained with 7 loading threads per server process. Hence, with two server processes per machine and 8 machines, 112 files were being read at the same time. Also notice that in a cluster architecture there is constant need for communication during loading, since every new URIs and literals must be encoded identically across the cluster; hence shared dictionaries must be accessed. Thus, a single loader thread counts for about 250% CPU across the cluster. The load was non-transactional and with no logging, to maximize performance. Aggregate load rates of up to 2.5M quads per second were observed for periods of up to 30 minutes. The total loading time for the dataset of 50 billion triples is about 6h 28 minutes, which makes the average loading speed 2.14M triples per second.

The largest load (150B quads) was slowed down by one machine showing markedly lower disk write throughput than the others. On the slowest machine iostat showed a continuous disk activity of about 700 device transactions per second, writing anything from 1 to 3 MB of data per second. On the other machines, disks were mostly idle with occasional flushing of database buffers to disk producing up to 2000 device transactions per second and 100MB/s write throughput. Since data is evenly divided and 2 of 16 processes were not runnable because the OS had too much buffered disk writes, this could stop the whole cluster for up to several minutes at a stretch. Our theory is that these problems were being caused by hardware malfunction.

To complete the 150B load, we interrupted the stalling server processes, moved the data directories to different drives, and resumed the loading again. The need for manual intervention, and the prior period of very slow progress makes it hard to calculate the total time it took for the 150B load.

### 1.4.3 Notes on the BI Workload

The test driver can run with single-user run or multi-user run.

- *Single user* run: This simulates the case that one user executes the query mix against the system under test.
- *Multi-user* run: This simulates the case that multiple users concurrently execute query mixes against the system under test.

All BSBM BI runs were with minimal disk IO. No specific warm-up was used and the single user run was run immediately following a cold start of the multi-user run. The working set of BSBM BI is approximately 3 bytes per quad in the database. The space consumption without literals and URI strings is 8 bytes with Virtuoso column store default settings. For a single user run, typical CPU utilization was around 190 of 256 core threads busy. For a multi-user run, all core threads were typically busy. Hence we see that the 4 user run takes roughly 3 times the real time of the single user run.

### 1.4.4 Benchmark Results

The following terms will be used in the tables representing the results.

- *Elapsed runtime* (seconds): the total runtime of all the queries excluding the time for warm-up runs.
- *Throughput*: the number of executed queries per hour.
  *Throughput = (Total # of executed queries) * (3600 / ElapsedTime) * scaleFactor*.
  Here, the scale factor for the 50 billion triples dataset and 150 billion triples dataset is 500 and 1500, respectively.
- *AQET*: Average Query Execution Time (seconds): The average execution time of each query computed by the total runtime of that query and the number of executions: *AQET(q) = (Total runtime of q) / (number of executions of q)*.

Some results seem noisy, for instance Q2@50B, Q4@50B, Q4@150B are significantly cheaper in the multi-client-setup. Given the fact that the benchmark was run in drill-down mode, this is unexpected. It could be countered by performing more runs, but, this would lead to very large run-times as the BI workload has many long-running queries.

In the following, we discuss the above performance results over several specific queries Q2 and Q3.

Query 2 in the BI use case:

```
SELECT ?otherProduct ?sameFeatures {
?otherProduct a bsbm:Product .
FILTER(?otherProduct != %Product%)
{ SELECT ?otherProduct (COUNT(?otherFeature) AS ?sameFeatures) {
   %Product% bsbm:productFeature ?feature .
   ?otherProduct bsbm:productFeature ?otherFeature .
```

| | 50 Billion triples | | 150Billion triples | |
|---|---|---|---|---|
| | Single-Client | 4-Clients | Single-Client | 4-Clients |
| runtime | 3733s | 9066s | 12649s | 29991s |
| Tput | 12.052K | 19.851K | 10.671K | 18.003K |
| | AQET | AQET | AQET | AQET |
| Q1 | 622.80s | 1085.82 | 914.39s | 1591.37s |
| Q2 | 189.85s | 30.18 | 196.01s | 507.02s |
| Q3 | 337.64s | 2574.65 | 942.97s | 8447.73s |
| Q4 | 18.13s | 6.3s | 183.00s | 125.71s |
| Q5 | 187.60s | 319.75s | 830.26s | 1342.08s |
| Q6 | 47.64s | 34.67s | 24.45s | 191.42s |
| Q7 | 36.96s | 39.37s | 58.63s | 94.82s |
| Q8 | 256.93s | 583.20s | 1030.73s | 1920.03s |

**Table 1.3** Business Intelligence Use Case: Detailed Results

```
    FILTER(?feature=?otherFeature)
  } GROUP BY ?otherProduct }}
ORDER BY DESC(?sameFeatures) ?otherProduct LIMIT 10
```

BSBM BI Q2 is a lookup for the products with the most features in common with a given product. The parameter choices (i.e., %Product%) produce a large variation in run times. Hence the percentage of the query's timeshare varies according to the repetitions of this query's execution. For the case of 4-clients, this query is executed for 4 times which can be the reason for the difference timeshare between single-client and 4-client of this query.

Query 3 in the BI use case:

```
 SELECT ?product
(xsd:float(?monthCount)/?monthBeforeCount AS ?ratio) { { SELECT
?product (COUNT(?review) AS ?monthCount) {
  ?review bsbm:reviewFor ?product .
  ?review dc:date ?date .
  FILTER(?date >= "%ConsecutiveMonth_1%"^^<http://www.w3.org/2001/XMLSchema#date>
      && ?date < "%ConsecutiveMonth_2%"^^<http://www.w3.org/2001/XMLSchema#date>) }
  GROUP BY ?product }
{ SELECT ?product (COUNT(?review) AS ?monthBeforeCount) {
  ?review bsbm:reviewFor ?product .
  ?review dc:date ?date .
  FILTER(?date >= "%ConsecutiveMonth_0%"^^<http://www.w3.org/2001/XMLSchema#date>
      && ?date < "%ConsecutiveMonth_1%"^^<http://www.w3.org/2001/XMLSchema#date>) }
  GROUP BY ?product
  HAVING (COUNT(?review)>0) }}
ORDER BY DESC(xsd:float(?monthCount) / ?monthBeforeCount) ?product
LIMIT 10
```

The query generates a large intermediate result: all the products and their review count on the latter of the two months. This takes about 16GB (in case of 150 billion triples), which causes this to be handled in the buffer pool, i.e. the data does not all have to be in memory. With multiple users connected to the same server process, there is a likelihood of multiple large intermediate results having to be stored at the same time. This causes the results to revert earlier to a representation that can overflow to disk. Supposing 3 concurrent instances of Q3 on the same server process, the buffer pool of approximately 80G has approximately 48G taken by these

intermediate results. This causes pages needed by the query to be paged out, leading to disk access later in the query. Thus the effect of many instances of Q3 on the same server at the same time decreases the throughput more than linearly. This is the reason for the difference in timeshare percentage between the single-user and multi-user runs. The further problem in this query is that the large aggregation on count is on the end result, which re-aggregates the aggregates produced by different worker threads. This re-aggregation is due to the large amount of groups quite costly; therefore it dominates the execution time: the query does not parallelize well. A better plan would hash-split the aggregates early, such that re-aggregation is not required.

| | 50 Billion triples | |
|---|---|---|
| | Single-Client | 4-Clients |
| runtime | 1988s | 4690s |
| Tput | 22.629K | 38.375K |
| | AQET | AQET |
| Q1 | 58.93 | 72.26 |
| Q2 | 2.15 | 20.14 |
| Q3 | 449.42 | 656.52 |
| Q4 | 36.35 | 75.09 |
| Q5 | 95.37 | 312.33 |
| Q6 | 0.31 | 25.85 |
| Q7 | 7.72 | 27.96 |
| Q8 | 154.47 | 292.77 |

**Table 1.4** Business Intelligence Use Case: Updated Results in March 2013

The benchmark results in the Table 1.3 are taken from our experiments running in January 2013. With more tuning in the Virtuoso software, we have re-run the benchmark with the dataset of 50B triples. The updated benchmark results in Table 1.4 show that the current version of Virtuoso software, namely Virtuoso7-March2013, can run the BSBM BI with a factor of 2 faster than the old version (i.e., the Virtuoso software in January). Similar improvement on the benchmark results is also expected when we re-run the benchmark with the dataset of 150B triples.

We now discuss the performance results in the Explore use case. We notice that these 4-client results seem more noisy than the single-client results and therefore it may be advisable in future benchmarking to also use multiple runs for multi-client tests. What is striking in the Explore results is that Q5 dominates execution time.

Query 5 in the Explore use case:

```
 SELECT DISTINCT ?product ?productLabel
WHERE {
 ?product rdfs:label ?productLabel .
 FILTER (%ProductXYZ% != ?product)
 %ProductXYZ% bsbm:productFeature ?prodFeature .
 ?product bsbm:productFeature ?prodFeature .

 %ProductXYZ% bsbm:productPropertyNumeric1 ?origProperty1 .
 ?product bsbm:productPropertyNumeric1 ?simProperty1 .
 FILTER (?simProperty1 < (?origProperty1 + 120) &&
```

| | 50 Billion triples | | 150Billion triples | |
|---|---|---|---|---|
| | Single-Client | 4-Clients | Single-Client | 4-Clients |
| runtime | 931s (100 runs) | 15s (1run) | 1894s (100 runs) | 29s (1 run) |
| Tput | 4.832M | 11.820M | 7.126M | 18.386M |
| | AQET | AQET | AQET | AQET |
| Q1 | 0.066s | 0.415s | 0.113s | 0.093s |
| Q2 | 0.045s | 0.041s | 0.066s | 0.086s |
| Q3 | 0.112s | 0.091s | 0.111s | 0.116s |
| Q4 | 0.156s | 0.102s | 0.308s | 0.230s |
| Q5 | 3.748s | 6.190s | 8.052s | 9.655s |
| Q7 | 0.155s | 0.043s | 0.258s | 0.360s |
| Q8 | 0.100s | 0.021s | 0.188s | 0.186s |
| Q9 | 0.011s | 0.010s | 0.011s | 0.011s |
| Q10 | 0.147s | 0.020s | 0.201s | 0.242s |
| Q11 | 0.005s | 0.004s | 0.006s | 0.006s |
| Q12 | 0.014s | 0.019s | 0.013s | 0.010s |

**Table 1.5** Explore Use Case: Detailed Results

```
        ?simProperty1 > (?origProperty1 - 120))

 %ProductXYZ% bsbm:productPropertyNumeric2 ?origProperty2 .
 ?product bsbm:productPropertyNumeric2 ?simProperty2 .
 FILTER (?simProperty2 < (?origProperty2 + 170) &&
        ?simProperty2 > (?origProperty2 - 170))
} ORDER BY ?productLabel LIMIT 5
```

Q5 asks for the 5 most similar products to one given product, based on two numeric product properties (using range selections). It is notable that such range selections might not be computable with the help of indexes; and/or the boundaries of both 120 and 170 below and above may lead to many products being considered 'similar'. Given the type of query, it is not surprising to see that Q5 is significantly more expensive than all other queries in the Explore use case (the other queries are lookups that are index computable. – this also means that execution time on them is low regardless of the scale factor). In the explore use case, most of the queries have the constant running time regardless of the scalefactor, thus computing the throughput by multiplying the qph (queries per hour) with the scalefactor may show a significant increase between the cases of 50-billion and 150-billion triples. In this case, instead of the throughput metric, it is better to use another metric, namely qmph (number of query mixes per hour).

| | Single Client | 4-Clients |
|---|---|---|
| 50B | 4253.157 | 2837.285 |
| 150B | 2090.574 | 1471.032 |

**Table 1.6** Explore Results: Query Mixes Per Hour

## 1.5 Emergent Schemas

In this section, we describe solutions for deriving an *emergent relational schema* from RDF triples, that one could liken to an UML class diagram. These solutions have been implemented in the RDF parser of the open-source research column store, MonetDB, which we call MonetDB/RDF. A more extensive description of this work can be found in [9].

Our problem description is as follows. Given a (very) large set of RDF triples, we are looking an emergent schema that describes this RDF data consisting of *classes* with their *attributes* and their *literal types*, and the *relationships* between classes for URI objects, but:
*(a)* the schema should be *compact*, hence the amount of classes, attributes and relationships should be as small as possible, such that it is easily understood by humans, and data does not get scattered over too many small tables.
*(b)* the schema should have *high coverage*, so the great majority of the triples in the dataset should represent an attribute value or relationship of a class. Some triples may not be represented by the schema (we call these the "non-regular" triples), but try to keep this loss of coverage small, e.g. $<10\%$.
*(c)* the schema should be *precise*, so the amount of *missing* properties for any subject that is member of such an recognized class is minimized.

Our solution is based on finding *Characteristic Sets* (CS) of properties that co-occur with the same subject. We obtain a more compact schema than [10], by using the TF/IDF (Term Frequency / Inverted Document Frequency) measure from information retrieval [15] to detect *discriminative* properties, and using semantic information to *merge* similar CS's. Further, a schema graph of CS's is created by analyzing the co-reference relationship statistics between CS's.

Given our intention to provide users an easy-to-understand emergent schema, our second challenge is to determine logical and short *labels* for the classes, attributes and relationships. For this we use ontology labels and class hierarchy information, if present, as well as CS co-reference statistics, to obtain class, attribute and relational labels.

### *1.5.1 Step1: Basic CS Discovery*

**Exploring CS's.** We first identify the basic set of CS's by making one pass through all triples in the SPO (Subject, Predicate, Object) table created after bulk-loading of all RDF triples. These basic CS's are secondly further split out into combinations of (property, literal-type), when the object is a literal value. Thus, for each basic CS found, we may have multiple CS variants, one for each combination of occurring literal types. We need the information on literal types because our end objective is RDF storage in relational tables, which allow only a single type per column.

**Exploring CS Relationships.** A foreign key (FK) relationship between two CS's happens when a URI property of one CS typically refers in the object field to mem-

bers of one other CS (object-subject references). Therefore, we make a second pass over all triples with a non-literal object, look up which basic CS the reference points, and count the frequencies of the various destination CS's.

### *1.5.2 Step2: Dimension Tables Detection*

There tends to be a long tail of infrequently occurring CS's, and as we want a compact schema, the non-frequent CS's should be pruned. However, a low-frequency CS which is referred to many times by high-frequency CS's in fact represents important information of the dataset and should be part of the schema. This is similar to a *dimension table* in a relational data warehouse, which may be small itself, but may be referred to by many millions of tuples in large fact tables, over a foreign key. However, detecting dimension tables should not be handled just based on the number of *direct* relationship references. The relational analogy here are *snowflake* schemas, where a finer-grained dimension table like NATION refers to an even smaller coarse-grained dimension table CONTINENT. To find the transitive relationships and their relative importance, we use the recursive PageRank[13] algorithm on the graph formed by all CS's (vertexes) and relationships (edges). As a final result, we mark low-frequency CS's with a high rank as "dimension" tables, which will protect them later from being pruned.

### *1.5.3 Step3: Human-friendly Labels*

When presenting humans with a UML or relational schema, short labels should be used as aliases for machine-readable and unique URIs for naming classes, attributes and relationships. For assigning labels to CS's, we exploit both structural and semantic information (ontologies).

**Type Properties.** Certain specific properties (e.g., rdf:type) explicitly specify the *class* or *concept* a subject belongs to. By analyzing the *frequency distribution* of different RDF type property values in the triples that belong to a CS, we can find a class label for the CS. As ontologies usually contain *hierarchies*, we create a histogram of type property values per CS that is aware of hierarchies. The type property value that describes most of the subjects in the CS, but is also as specific as possible is chosen as the URI of the class. If a ontology class URI is found, we can use its label as the CS's label. In Figure 1.3, the value "Ship" is chosen.

**Ontologies.** Even if no type property is present in the CS, we can still try to match a CS to an ontology class. We compare the property set of the CS with the property sets of ontology classes using the TF/IDF similarity score [15]. This method relies on identifying "discriminative" properties, that appear in few ontology classes only, and whose occurrence in triple data thus gives a strong hint for the membership of a specific class. An example is shown in Figure 1.2.

The ontology class correspondence of a CS, if found, is also used to find labels for properties of the CS (both for relationships and literal properties).

**Relationships between CS's.** If the previous approaches do not apply, we can look at which other CS's refer to a CS, and then use the URI of the referring property to derive a label. For example, a CS that is referred as <*author*> indicates that this CS represents instances of a <*Author*> class. We use the most frequent relationship to provide a CS label. Figure 1.4 shows an example of such "foreign key" names.

| $CS_2$ |
|---|
| rdf:type |
| gor:validFrom |
| gor:validThrough |
| gor:hasCurrency |
| gor:hasCurrencyValue |
| gor:hasUnitOfMeasurement |
| gor:valueAddedTaxIncluded |
| gor:eligibleTransactionVolume |

*(prefix gor: http://purl.org/goodrelations/v1#)*

| PriceSpecification |
|---|
| gor:description |
| gor:name |
| gor:eligibleTransactionVolume |
| gor:validFrom |
| gor:validThrough |
| gor:hasCurrency |
| gor:hasCurrencyValue |
| gor:hasUnitOfMeasurement |
| gor:valueAddedTaxIncluded |
| gor:hasMaxCurrencyValue |
| gor:hasMinCurrencyValue |

**Fig. 1.2** Example CS vs. Ontology Class

| Level | Type | % |
|---|---|---|
| 0 | Thing | 100 |
| 1 | MeanOfTransportation | 100 |
| 2 | Ship | 97 |
| 2 | Automobile | 2 |
| 2 | SpaceShuttle | 1 |

**Fig. 1.3** CS Type Property values

| FK name | #CS | #tuples |
|---|---|---|
| instrument | 3 | 93532 |
| author | 1 | 5 |

**Fig. 1.4** References to a CS

**URI shortening.** If the above solutions cannot provide us a link to ontology information, for providing attribute and relationship labels we resort to a practical fallback, based on the observation that often property URI values do convey a hint of the semantics. That is, for finding labels of CS properties we shorten URIs (e.g., `http://purl.org/goodrelations/v1#offers` becomes `offers`), by removing the ontology prefix (e.g., `http://purl.org/goodrelations/v1#`) or simply using the part after the last slash, as suggested by [11].

### *1.5.4 Step4: CS Merging*

To have a compact schema, we further reduce the number of tables in the emergent relational schema by merging CS's, using either *semantic* or *structural* information.
**Semantic Merging.** We can merge two CS's on semantic grounds when both CS class labels that we found were based on ontology information. Obviously, two CS's whose label was created using the same ontology class URI represent the same concept, and thus can be merged. If the labels stem from *different* ontology classes we can observe the subclass hierarchy and identify the common concept/class shared by both CS's (e.g.,<*Athlete*> is a common class for <*BasketballPlayer*> and <*BaseballPlayer*>), if any, and then justify whether these CS's are similar based on the "generality" of the concept. Here the "generality" score of a concept is computed by the percentage of instances covered by it and its subclasses among all the

instances covered by that ontology. Two CS's whose labels share a non-general ancestor in an ontology class hierarchy can be merged.

**Structural Merging.** The structural similarity between two CS's can be assessed by using the set of properties in each CS and the found relationships to them with other CS's. As original class can be identified based on "discriminating" properties (based on TF/IDF scoring), we merge two CS if their property sets have a high TF/IDF similarity score. Additionally, as a subject typically refers to only one specific entity via a property, we also merge two CS's which are both referred from the same CS via the same property.

### *1.5.5 Step5: Schema and Instance Filtering*

We now perform final post-processing to clean up and optimize both the schema and the data instances in it. At part of this phase, all RDF triples are visited again, and either become stored in relational tables (typically >90% of the triples, which we consider regular), and the remainder gets stored separately in a PSO table. Hence, our final result is a set of relational tables with foreign keys between them, and a single triple table in PSO format.

**Filtering small tables.** After the merging process, most of these merged classes (i.e., surviving merged CS's) cover a large amount of triples. However, it may happen that some classes still cover a limited number of RDF subjects, (i.e. less than 0.1% of all data). As removing these classes will only marginally reduce coverage, we remove them from the schema (except classes that were recognized as dimension tables with the described PageRank method). All triples of subjects belonging to these classes will be moved to the separate PSO table.

**Maximizing type homogeneity.** Literal object values corresponding to each attribute in a class can have several different types e.g., number, string, dateTime, etc. The relational model can only store a single type in each column, so in case of type diversity multiple columns will be used for a single property. As the number of columns can be large just due to a few triples having the wrong type (dirty data), we minimize this number by filtering out all the infrequent literal types (types that appear in less than 5% of all object values) for each property. The triples with infrequent literal types are moved to the separate PSO table.

**Minimizing the number of infrequent columns.** Infrequent columns are those that have lots of NULL values. If the property coverage is less than a certain threshold value (i.e., 5%), that property is infrequent and all the RDF triples of that property are treated as irregular data and moved to the separate PSO table.

**Filtering the relationships.** We further filter out *infrequent* or "dirty" relationships between classes. A relationship between $cs_i$ and $cs_j$ is infrequent if the number of references from $cs_i$ to $cs_j$ is much smaller than the the frequency of $cs_i$ (e.g., less than 1% of the CS's frequency). A relationship is considered dirty if most but not all the object values of the referring class (e.g., $cs_i$) refer to the instances of the referred

class ($cs_j$). In the former case, we simply remove the relationship information between two classes. In the latter case, the triples in $cs_i$ that do not refer to $cs_j$ will be filtered out (placed in the separate PSO table).

**Multi-valued attributes.** The same subject may have 0, 1 or even multiple triples with the same property, which in our schema leads to an attribute with cardinality $> 1$. While this is allowed in UML, direct storage of such values is not possible in relational databases. Practitioners handle this by creating a separate table that contains the primary key (subject oid) and the value (which given literal type diversity may be multiple columns). The MonetDB/RDF system does this, but only creates such separate storage if really necessary. That is, we analyze the mean number of object values (*meanp*) per property. If the *meanp* of a property $p$ is not much greater than 1 (e.g., less than 1.1), we consider $p$ as a single-valued property and only keep the first value of that property while moving all the triples with other object values of this property to the non-structural part of the RDF dataset. Otherwise, we will add a table for storing all the object values of each multi-valued property.

### *1.5.6 Final Schema Evaluation*

For evaluating the quality of the final schema, we have conducted extensive experiments over a wide range of real-world and synthetic datasets (i.e., DBpedia[4], PubMed[5], DBLP[6], MusicBrainz[7], EuroStat[8], BSBM[9], SP2B[10], LUBM[11] and WebDataCommons[12]). The experimental results in Table 1.7 show that we can derive a compact schema from each dataset with a relative small number of tables. We see that the synthetic RDF benchmark data (BSBM, SP2B, LUBM) is fully relational, and also all dataset with non-RDF roots (PubMed, MusicBrainz, EuroStat) get >99% coverage. Most surprisingly, the RDFa data that dominates WebDataCommons and even DBpedia are more than 90% regular.

**Labeling Evaluation.** We further evaluate the quality of the labels in the final schema by showing the schema of DBpedia and WebDataCommons (complex and, may be, "dirty" datasets) to 19 humans. The survey asking for rating label quality with the 5-point Likert scale from 1 (bad) to 5 (excellent) shows that 78% (WebDataCommons) and 90% (DBpedia) of the labels are rated with 4 points (i.e., "good") or better.

---

[4] http://dbpedia.org - we used v3.9

[5] http://www.ncbi.nlm.nih.gov/pubmed

[6] http://gaia.infor.uva.es/hdt/dblp-2012-11-28.hdt.gz

[7] http://linkedbrainz.c4dmpresents.org/data/musicbrainz_ngs_dump.rdf.ttl.gz

[8] http://eurostat.linked-statistics.org

[9] http://wifo5-03.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark/

[10] http://dbis.informatik.uni-freiburg.de/forschung/projekte/SP2B/

[11] http://swat.cse.lehigh.edu/projects/lubm/

[12] A 100M triple file of http://webdatacommons.org

**Table 1.7** Number of tables and coverage percentage after merging & filtering steps

| Datasets | Number of tables | | | | Coverage – Metric $C$ (%) | | |
|---|---|---|---|---|---|---|---|
| | before merging | | after | remove | remove | prune | final |
| | basic CS's | frequent CS's | merging | small tables | small tables | infrequent properties | schema |
| Pubmed | 3340 | 1754 | 14 | 10 | 99.99 | 99.74 | **99.72** |
| DBpedia | 472244 | 213851 | 517 | 298 | 94.12 | 91.73 | **90.82** |
| BSBM | 51 | 51 | 8 | 8 | 100 | 100 | **100.00** |
| DBLP | 251 | 181 | 9 | 6 | 99.99 | 99.68 | **99.60** |
| SP2B | 554 | 410 | 13 | 9 | 99.99 | 99.65 | **99.65** |
| MusicBrainz | 27 | 27 | 12 | 12 | 100 | 99.9 | **99.19** |
| LUBM | 17 | 16 | 12 | 11 | 100 | 100 | **100.00** |
| WebDataCommons | 13913 | 8319 | 780 | 113 | 98.79 | 94.55 | **92.90** |
| EuroStat | 44 | 27 | 5 | 5 | 99.51 | 99.32 | **99.32** |

**Computational cost & Compression.** Our experiments also show that the time for detecting the emerging schema is negligible comparing to bulk-loading time for building a single SPO table, and thus the schema detection process can be integrated into the bulk-loading process without any recognizable delay. Additionally, the database size stored using relational tables can be 2x smaller than the database size of a single SPO triple table since in the relational representation the S and P columns effectively get compressed away and only the O columns remain.

**Final words.** We think the emergent schema detection approach we developed and evaluated is promising. The fact that all tested RDF datasets turned out highly regular, and that good labels for them could be found already provides immediate value, since MonetDB/RDF can now simply be used to load RDF data in a SQL system; hence existing SQL applications can now be leveraged on RDF without change. We expect that all systems that can store both RDF and relational data (this includes besides Virtuoso also the RDF solutions by Oracle and IBM) could incorporate the possibility to load RDF data and query it both from SQL and SPARQL.

Future research is to verify the approach on more RDF dataset and further tune the recognition algorithms. Also, the second and natural step is now to make the SPARQL engine aware of the emergent schema, such that its query optimization can become more reliable and query execution can reduce the join effort in evaluating so-called SPARQL star-patterns. In benchmarks like LUBM and BSBM our results show that SPARQL systems could become just as fast as SQL systems, but even on "real" RDF datasets like DBpedia 90% of join effort can likely be accelerated. Work is underway to verify this both in MonetDB and Virtuoso.

## References

1. IBM DB2. www.ibm.com/software/data/db2/.
2. MonetDB column store. https://www.monetdb.org/.
3. Openlink Software Blog. http://www.openlinksw.com/weblog/oerling/.

4. Daniel J. Abadi. Query execution in column-oriented database systems. MIT PhD Dissertation, 2008. PhD Thesis.
5. Christian Bizer and Andreas Schultz. The berlin sparql benchmark. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 5(2):1–24, 2009.
6. Peter Boncz, Thomas Neumann, and Orri Erling. Tpc-h analyzed: Hidden messages and lessons learned from an influential benchmark. TPCTC, 2013.
7. Orri Erling. Virtuoso, a hybrid rdbms/graph column store. *IEEE Data Eng. Bull*, 35(1):3–8, 2012.
8. Andrew Lamb et al. The vertica analytic database: C-store 7 years later. *Proceedings of the VLDB Endowment*, pages 1790–1801, 2012.
9. P. Minh-Duc et al. Deriving an emergent relational schema from rdf data. In *ISWC*. (submitted), 2014.
10. T. Neumann et al. Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. In *ICDE*, 2011.
11. Robert Neumayer, Krisztian Balog, and Kjetil Nørvåg. When simple is (more than) good enough: Effective semantic search with (almost) no semantics. In *Advances in Information Retrieval*, pages 540–543. Springer, 2012.
12. Pat O'Neil, Elizabeth J O'Neil, and Xuedong Chen. The star schema benchmark (ssb). *Pat*, 2007.
13. L. Page et al. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.
14. Minh-Duc Pham. Self-organizing structured RDF in monetdb. In *ICDE Workshops*, pages 310–313, 2013.
15. Gerard Salton and Michael J McGill. Introduction to modern information retrieval. 1983.
16. Petros Tsialiamanis et al. Heuristics-based query optimisation for sparql. In *EDBT*, pages 324–335, 2012.
17. Marcin Zukowski and Peter A Boncz. Vectorwise: Beyond column stores. *IEEE Data Eng. Bull.*, pages 21–27, 2012.